Computational Geometry

Prof. Sandeep Sen

Department of Computer Science and Engineering

Indian Institute of Technology, Delhi

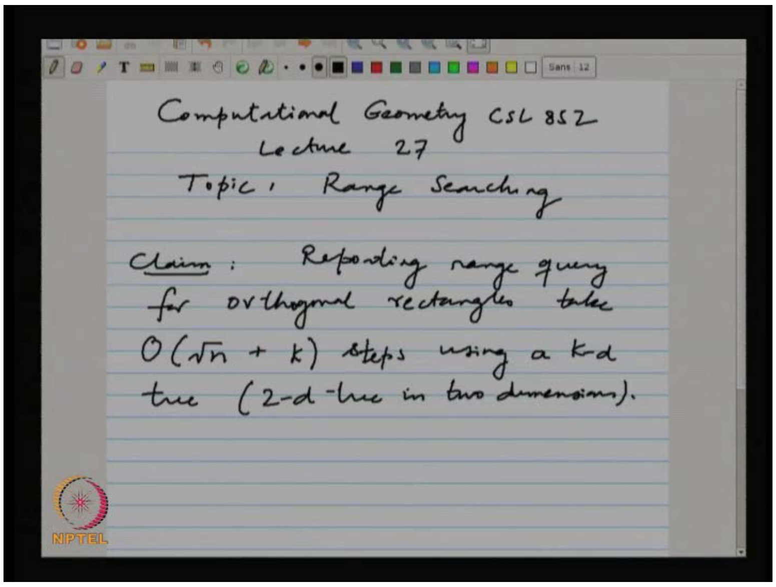
Module No. # 11

Range searching

Lecture No. # 02

Orthogonal Range Searching

(Refer Slide Time: 00:26)



Computational Geometry CSL 852
Lecture 27
Topic: Range Searching

Claim: Reporting range query for orthogonal rectangles take $O(\sqrt{n} + k)$ steps using a k-d tree (2-d-tree in two dimensions).

Recap; I think we left off where we are just about getting down to the analysis of the rectangular orthogonal orthogonal rectangular range searching in two dimensions, right. So, where did we leave off? We left off at a point.

(No audio from 00:44 to 00:57)

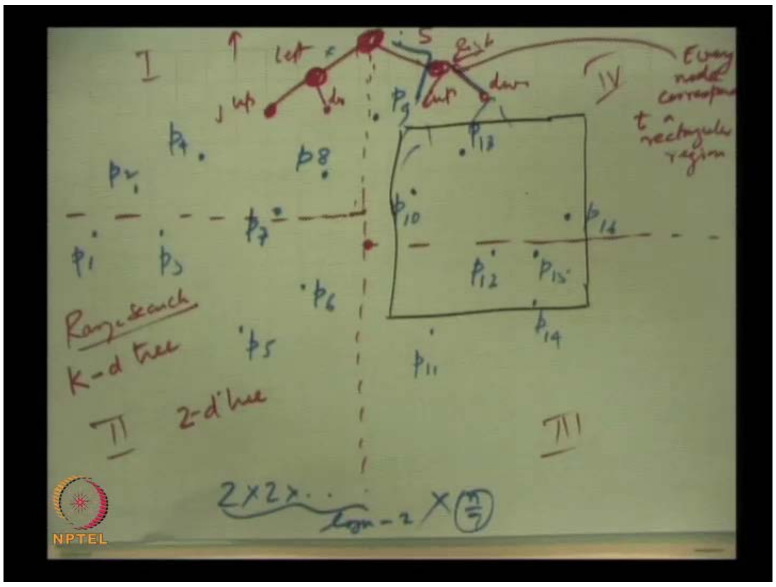
Right, right. So, I think I just proposed the result or just advertise the result; I did not really prove it, right. So, reporting range query for orthogonal rectangles,

(No audio from 01:27 to 01:48) where the 2D tree in two dimensions, right.

(No audio from 01:51 to 02:04)

So, any question about the basic scheme before we do the analysis? The Query tree; yeah, it is very simple. So, just we recap.

(Refer Slide Time: 02:16)

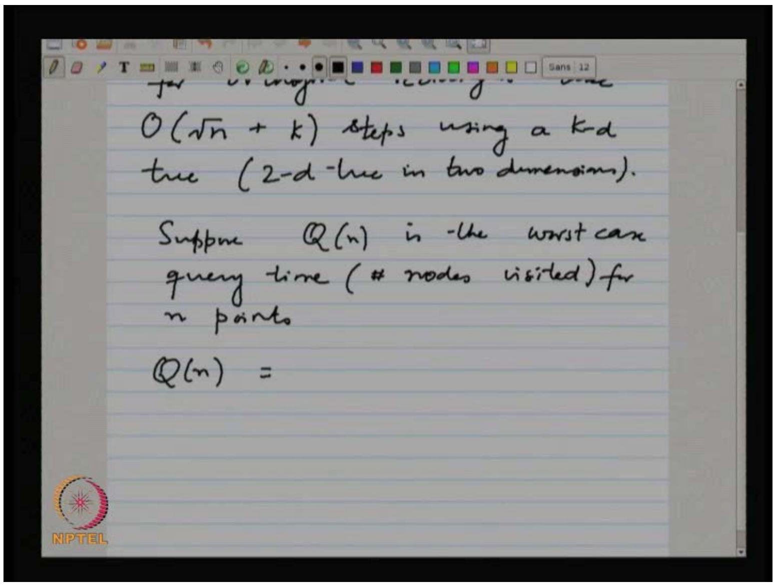


You know what did we do alternately is sort of sliced the plane using a median on the x co-ordinates and then median on the y co-ordinates and we obtained, you know smaller and smaller rectangles and then, when we query design when the query rectangle came along, we only found out… So, the tree was defined by these partitions; so, this, the root of the tree defines the entire plane or basically a bounded box which contains all the points. And the two children correspond to the left and the right half, then the children of that basically correspond to the up and down and then from this point onwards, recursively we build the data structure for the 4 quadrants, whatever the 4 partitions that we have obtained so far.

And then, when the query rectangle comes along, it starts from the root and then it take care… First, query the simple cases; does it complete completely contain the region defined by the by the node? If so, then you stop and repot all the points; you know, does it have does it basically have no overlap with the region? Then again, you just again end your range search, saying that…; No, sorry there are no points to before report it; the only case, where you actually continue down is when there is some kind of partial overlap with the regions, right.

So, that is how this range query proceeds. And, so, how do we do the analysis? You can actually, because the search tree is itself defined recursively; you can try to write a recurrence for the query time, right.

(Refer Slide Time: 03:48)


[illegible]

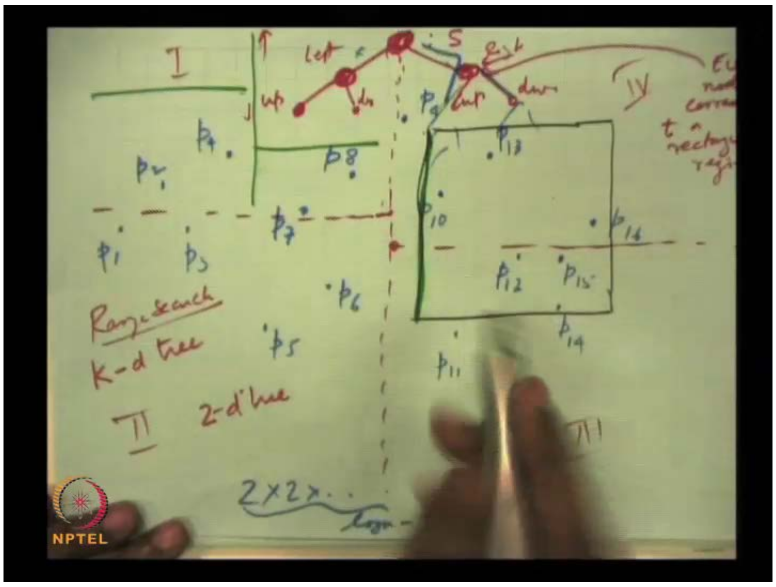
$O(\sqrt{n} + k)$ steps using a k-d tree (2-d-tree in two dimensions).

Suppose $Q(n)$ is the worst case query time (# nodes visited) for n points

$Q(n) =$

So, suppose, Q of n is the worst case query time. So, query time in this case, query time basically means the number of nodes that we visit along the way till the search terminates completely, right; number of nodes visited because every time we visit a node, we basically doing some constant kind of operations to find out whether it is a partial overlap, no overlap or complete overlap, right; and we basically want to write Q’s of n as something, right. So, the rectangle comes along. So, what we will do is, to simplify the analysis, we will actually look at in the following way:

(Refer Slide Time: 04:53)



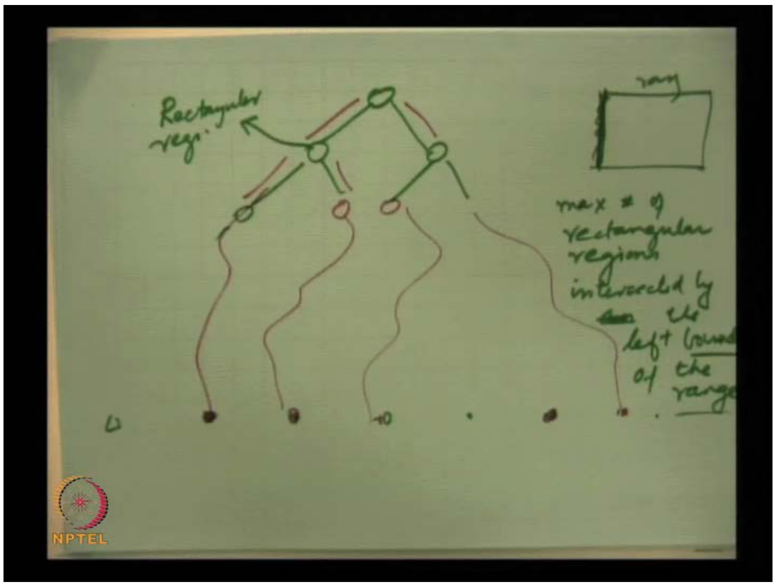
You look at the four boundaries of the rectangle; the two vertical boundaries and the two horizontal boundaries.

I will actually compute the number of intersections with a vertical line, analogously, horizontal line with the number of regions in the tree. See, this partitioning basically, I did not continue further, but essentially within each of these, you again have regions, right. So, you are basically subdividing into finer and finer rectangular regions.

So, I am going to compute the number of times, let us say, this edge; not number of times, but the number of regions that this edge intersects with. And then, analogously, this will not this will also intersect with the maximum number of that many regions. So, if I multiply the number of intersecting regions of this vertical boundary, I will multiply that by 4 and that will be give me the maximum number of nodes visited. Do you agree that? Let us try again.

So, I am trying to find out how many... So, as we proceed along this search tree, it could actually from a node, actually visit both its children; it is possible, right. So, in the end, you know we are going to basically look at what is the total number of… If I look at the tree, just the abstract tree.

(Refer Slide Time: 06:29)


Rectangular reg.
range
max # of rectangular regions intersected by the left bound of the range

This is my query tree; this is green tree, not a red tree. So, in the end, you know these are the leaf nodes. As we move down, it is possible that the rectangle could actually intersect both these and both of these, but what we are trying to analyze is that, eventually it is not going to visit all the leaf nodes, but you know it will visit only maximum of some subset of these leaf nodes, right. Because, some search path will end here, some search path will end here and so on and so forth, right.

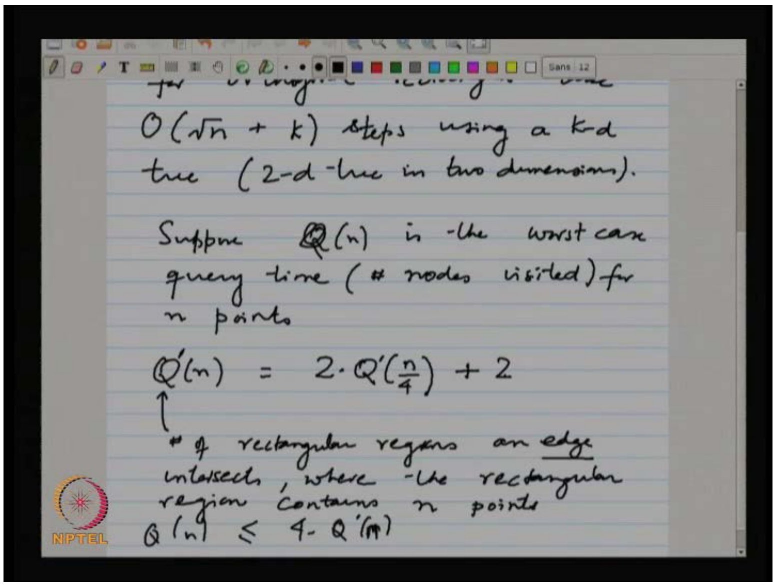
And, to find out how many nodes we visit, I will try to analyze that using… Not looking at the entire rectangle, but looking at an edge of the rectangle. So, the maximum and each of these correspond to rectangular region; some rectangular region that corresponds to this, some rectangular region. Each node is a rectangular region. So, maximum number of rectangular regions intersected by an, let us say, this edge by the left boundary; do not confuse between the range and these rectangular regions; this is the edge left boundary of the range, the rectangular region.

So, I will find out how many of these rectangular regions or in other words, how many of these nodes, this left edge can visit? And then, I multiply by that 4; that will be the maximum number of nodes this rectangle can visit in the course of the search process.

You are trying to find out, how many of these rectangular regions, this range can overlap with? It means that at least one of the edges should be overlapping with that.

So, now, I am trying to find out for an each edge, what is the maximum number of rectangular regions that this edge can intersect? And in the end, if I multiply that by 4, that will certainly be an upper bound and the number of nodes, this range can actually visit; at least, from the edges must be intersecting the rectangular region; otherwise, there is no overlap. We are counting the number of partial overlap; so, I am counting the number of partial overlaps by first looking at a single edge and I multiply that like before.

(Refer Slide Time: 09:30)



$O(\sqrt{n} + k)$ steps using a k-d tree (2-d-tree in two dimensions).

Suppose $Q(n)$ is the worst case query time (# nodes visited) for n points

$Q'(n) = 2 \cdot Q'(\frac{n}{4}) + 2$

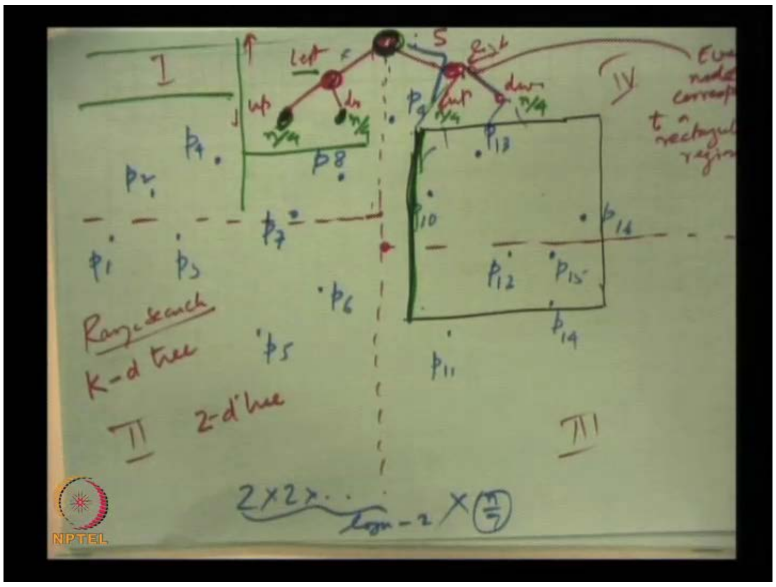
# of rectangular regions an edge intersects, where the rectangular region contains n points

$Q(n) \leq 4 \cdot Q'(n)$

So, when I am writing this query this recurrence, this recurrence actually will correspond to the number of rectangular regions an edge intersects, where the rectangular region contains n points. So, that is what Q’s of n denotes.

So, the actual number of nodes that we can visit during the course of the range query is this Q’s of n; Maybe, I should call it Q prime, to distinguish it from this Q’s of n; this is for the rectangle, this is for an edge of the rectangle. And so, the claim is that Q’s of n will be no more than four times Q prime of n, right. So, I will focus on writing a recurrence for the Q prime of n and I Claim that this is this can be bounded by two times Q prime of n over 4 plus 2. Can we justify this? Let me put down the figure again.

(Refer Slide Time: 11:12)


Range search
K-d tree
2-d tree
I
II
III
IV
left
up
dn
S
down
n/4
Every node corresponds to a rectangular region
$2\times2\times\ldots \log n-2 \times (n/4)$
NPTEL

So, let us say we are talking about this particular edge, right. Now, it will you know, this recurrence depends on how this tree is defined, whether the first node bisects the x co-ordinates or bisects the y co-ordinates. Let us assume that this one, the root node, bisects the… Well! In this case, it bisects the x co-ordinates; so, fine. Now, it bisects the x co-ordinates; I am talking about this edge and there is some overall boundary of this region that contains n points.

So, as we proceed down the tree, when we get to this level, that is we skip this level and go to the next level, we are back again to the same situation that these nodes are bisectors of x; this is bisector of x, this is bisector of x; these nodes are bisectors of y.

So, when we proceed down to this level, again they correspond to rectangular regions we shall at most how many points?

n over ? 4, right?

Because, we split twice; we split along y and then back to x, right. So, split it once; these are n over two, n over two and then again these are split again into two halves. So, these correspond to n over four nodes, each. So, these rectangular region correspond to n over four nodes, n over four points.

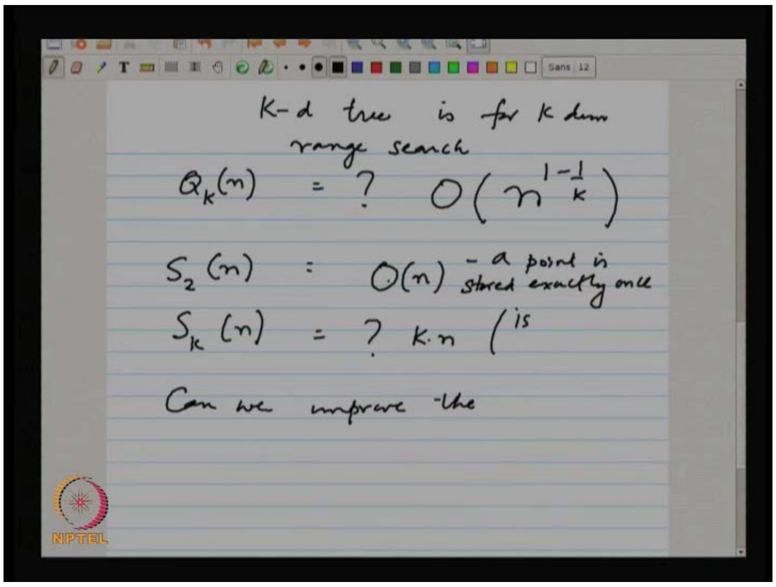
So, how many of these regions can this intersect? The left edge. You see that? So, it is going to be added to the left or the right of this vertical line. So, it is going to intersect at most two of these regions, which are n over four points; and plus of course, I have already two intersections because it intersects this rectangular region and this rectangular region; that is why I have added 2. So, I have added two because this edge intersects region four and region three plus recursively whatever it intersects within this region, that is two times Q prime of n over 4.

And so, what is the solution of this? Are you convinced about (( )). So, now, I just solve this; solve it later may be. So I claim this is basically is what defined. The additive constant is not going to make much more difference and you have to write the basic difference also. So, that is how you get squared of n.

(No audio from 14:12 to 14:32)

Fine. So, that is basically your 2D trees and in general, may be you want to guess again. What do you think will be result if you talk about k dimensions?

(Refer Slide Time: 14:43)



K-d tree is for k dim range search

$Q_k(n) = ?\ O(n^{1-\frac{1}{k}})$

$S_2(n) : O(n)$ - a point is stored exactly once

$S_k(n) = ?\ k.n$ (is

Can we improve the

So, the k-d tree is for k dimensional range search. So, query time in, let us say k dimension for n points, what do you think it should be? For two, we know the answer to be square root of n. Well! An exercise for you again. Sorry, k not t . So, you may just want to just test your understanding by writing the recurrence and solving it. And so, but, then if you observe this bound, you know, as k gets bigger, this actually becomes closer and closer to n, right. k becomes bigger; basically it becomes close to n. How much is the space? let us say, space for two dimensional k-d trees. So, in the way we are answering the query, what is implicit? How do you actually answer the query? depends on that the space will depend on that.

So, are we going to store all the points in each of the internal nodes? Yes or No? You do not need to, right. The moment we discover that, you know, suppose this entire node is contained within this region, we immediately we can maintain pointers, where to all the leaf nodes of the tree routed at this node; you simply list them out, that is all.

So, I do not need to store explicitly with each node, the points associated with that rectangular region. The moment I know that this rectangle contains this region, then I am going to list out all the points that are with the leaf nodes of the tree routed here. So, each point is stored exactly once, actually; it is very a powerful property. So, the query tree has a property that a point is stored exactly once; it is not even order of once, it is exactly once.

So, but there are some pointers; of course, the tree has a pointer structure because the primary tree has a structure and then for each of these nodes, we maintain certain pointers, but as far as points are concerned, they are stored exactly once. So, I will just write order n for two dimensional and what do you think about k dimensional?

What changes? k by 2? No. How can this space be sub-linear? The space has to be at least the number of points, right. Even k times n…

(No audio from 17:54 to 18:05)

What I will let you do is figure out how actually you are going to design this k-d tree and also, may be there is some kind of recursion that you need to understand before you can actually guess what this is. But yes, it is not going to be more than linear, whether it is k times n or order n, that is something I want to figure out. Is it k n or order n?

So, the question is that is a points… So, here a point is stored exactly once.

(No audio from 18:33 to 18:41)

Do you need to point store a point, more than once? I mean, the basic methodology that we have developed; Why should we store a point more than once? I mean, if you follow the same static that a… Let us say, in three dimensional, Q may be completely contained within and I am just simply going to list out something, but then there is a catch that, you know, we are dealing with higher dimensions, so, maybe there will be some kind of nested structure within each of these nodes. So, there is this something for you to consider.

It is not immediately obvious or maybe not, right. Maybe, just a simple structure; same thing; if there a partial overlap, we go and search this entire structure and if there is no overlap, you do not search that sub tree; if it is complete overlap, we list out all the nodes in the sub tree all the all the points in the sub tree.

So, just give it some thought whether it should be exactly once or it is something more complicated. No no, I am not ignoring with that way; I am just saying that the points are stored exactly once and certainly, the number of point has do not exceed overhead because the tree has size, you know, at most n leafs and each node is going to have a couple of pointers.

See, the points are something that could occupy a lot of storage depending on what you store with the points.

(Audio not clear from 20:10 to 20:19)

No, but each node will have only constant number of pointers, right and if there are order n number of nodes, will be order n number of pointers.

(Audio not clear from 20:28 to 20:39)

So, I do it does not matter, in which order I list out the points; I know that if this node is completely contained in the rectangle, all the nodes in the sub tree have to be listed; it does not matter in which order we list them; there is no obligation or to list them out in certain order .

(Audio not clear from 20:57 to 21:08)

Every leaf node has size k. Why do you think every leaf node has size k? It is k dimensional; the points have k co-ordinates, that is all. See, the points here… So, here is where, basically, now what we assume that we do not care about the number of bits etcetera whatever. So, it point to the point, real number; that is all.

So, it is not… it is just a models. Certainly, you know does not take care of bits and things like that. We are looking at basically a real number as a real number, that is all. So, this is a deficiency of this entire model, but then we have live with it and we are going to sort of (()). We are not going to divert from this point.

So, the question is, each point is data point, is stored exactly once, that is all. How much space each point occupies? That, we do not know; what kind of storage we are assigning to a real number. So, and it is only under this assumptions; so, yesterday I stated a result and I was not where is I am not very precise about it. So, I said that this square root of n bound is a best possible for order n space; actually, this is not order n space. The square root of n bound is a best possible, if you store a point exactly once.

If you. So, order n space, you can actually get better bounds. So, I will leave that as an exercise or I will properly formulate that when you should be able to… As we go on, we will cover more materials; will see how you can improve that. Keeping the space big of n, but there is no guarantee that a point to be stored exactly once; a point may be stored more than once, maybe three times, four times, ten times.

So, the pointers are only associated with the data structure. The points are the input data points. So, there is a distinction between the nodes, the pointers and the data points; the data points are real numbers and these are, you know, whatever your words or, you know, nodes, pointers.

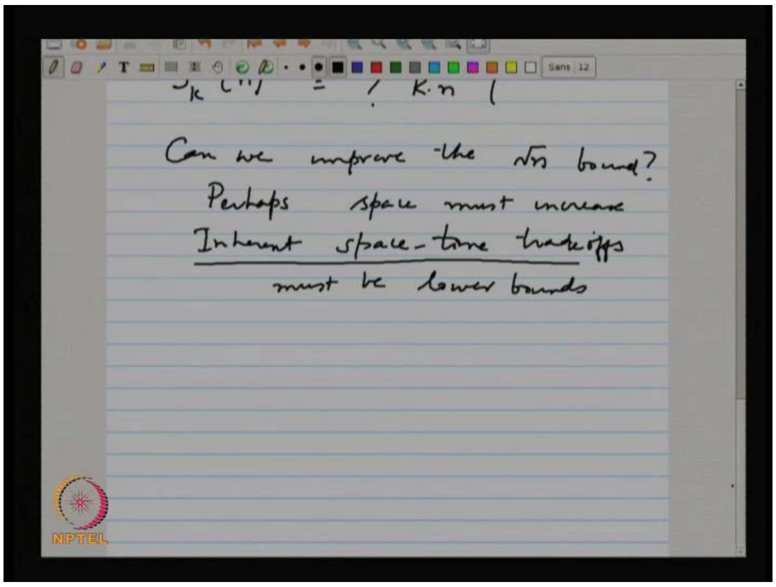
So, the obvious question here is that can we improve…So, the space is the order n, let us say, even if a point is stored more than once or some constant number of times, it is still order of n, o of n. So, we cannot do much better than in terms of space. In fact, in the case of k-d trees, we have stored a point exactly once; we cannot do anything better than that, but can we improve on the query time, basically. So, the square root of n plus, right, so, it is square root of n. So, square root of n is the number of nodes visited; let me go back again, square root of n is the number of nodes visited and of course, we also have to report all the points.

So, I may not proceed further along a node because this node is completely contained inside, but I have to still list out all the points. And so, and the points are being listed exactly once. So, therefore, the overall reporting time, the overall report time is square root of n plus k where k… I should not use k, because I am using it somewhere else. So, let us say, m, where m is the output size. So, also a point is reported exactly once because the nodes are actually disjoint, the regions are completely disjoint. So, I am reporting a point exactly once, only when we conclude that here this entire region is contained within the range.

So, square root of n is the number of nodes listed of the query tree and each point is reported exactly once. So, the total running time query time is square root of n plus n. The second part, you do not you cannot do anything about because that is the output size.

So, the question is can we improve on this? The square root of n does appear to be quite large. Exactly once, this is the lower bound, yes, but, you know, I may be able to use, you know, some big O of n space. Why should I be restricted to using storing a point exactly once? Maybe, I want to store it twice or thrice or four times or ten times. So, this bound does not apply when it is stored more than once. So, can we improve on the square root of n?

(Refer Slide Time: 25:39)



$\,_k(\ldots) = \;?\; k \cdot n\;($

Can we improve the $\sqrt{n}$ bound?

Perhaps space must increase

Inherent space-time tradeoffs

must be lower bounds

It is also possible that to improve on the square root of n bound, there may be some kind of a tradeoff in space, perhaps space has to increase. So, in general, intuitively, you can now, you can we have actually seen some examples in this course itself that if you want to speed up your queries, you may actually have a rather wasteful data structure. Think about, you know, you remember, we did this ray shooting, you know, where we had the segments, you know, we chopped them into trapezoids and we drew this vertical slabs and we notice that, you know, we can do this kind of ray shooting or point location using just two binary searches; once along find out which slab contains this and within the slab, where it is contained.

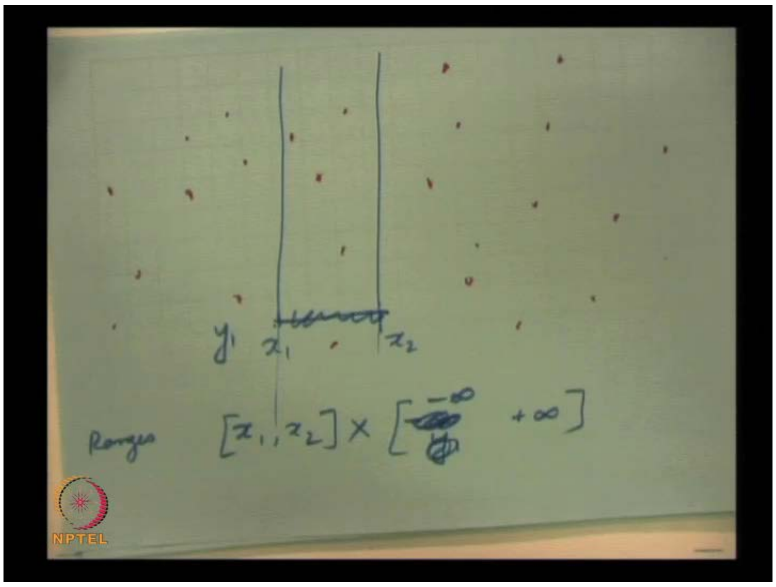
So, that was exactly like 2 log n, right, 2 binary searches. Problem was the storage, the storage was being growing up to some like quadratic because we had other mechanisms to improve the storage. So, but, intuitively it should be clear that, you know, if we allow more storage, you know, it may be possible to actually improve the query time. So, there is an inherent…

Now, sometimes, there is an inherent space time tradeoff, but these are very difficult to establish because we are talking here about lower bounds, right. It only make sense to say that… So, there may be some kind of inherent space time tradeoff, but the space time tradeoff must be some kind of a lower bound; must be lower bounds. Only then, you can claim that if I use less than, you know, five times n space, I have to use at least so much of query; I need atleast query time will be at least so much.

So, so, again these are rather hard, but then there are actually, you know, some some, really very deep and very beautiful results, which we are not going to get into, but I will (()) mention it on the way what kind of bounds are known.

So, right now, let me let me look at the slide variation of this two dimensional range query problem; I will relax the problem and develop a slightly different method.

(Refer Slide Time: 28:23)



So, instead of rectangles, let us look at some kind of slabs. So, here are points and my queries are are vertical slabs; vertical slabs essentially means that the ranges are of the form; so, the family of ranges is x 1 x 2 Cartesian product with minus infinity plus infinity.

So, in the y direction, you know, I am willing to come to look at the entire slab. Of course, this minus infinity plus infinity is actually bounded because the points are enclosed within a box. So, minus infinity plus infinity only means that the extent of the the the y the y coordinates.

So, if I pose this kind of a range query problem, how would you solve it? So, it becomes a one dimensional problem; basically, what we you saying is that. Well! this problem is effectively one dimensional problem. Because I can project all the points in the x direction and simply I will use my earlier solution, optimal solution, login query time or n space n log n.

So, let me then change the make the problem little bit more difficult. So, how about a three sided kind of a query, right. So, it is bounded on three sides, but not on the fourth side.

(No audio from 30:20 to 30:33)

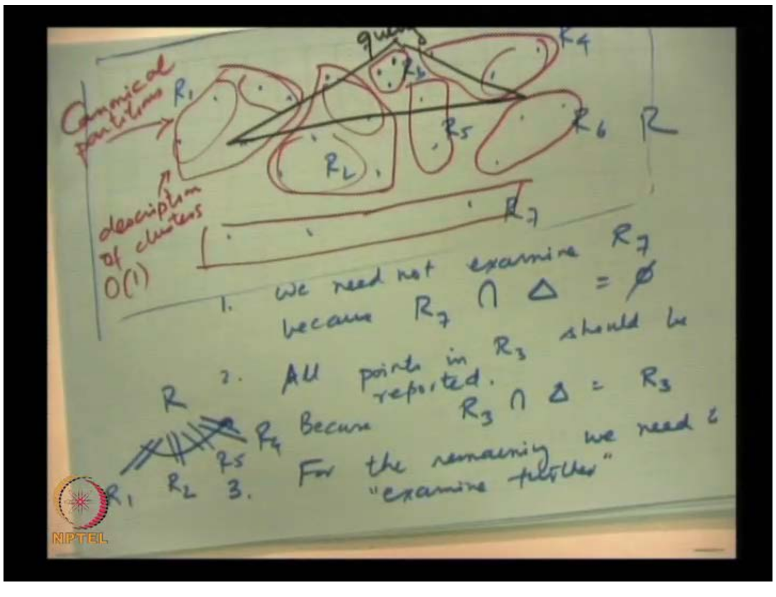
You can have two trees and then how do you find the intersection? The moment you have 2 trees; so, that I will I will tell you. So, in general, the problem with that kind of approach is that you have you are going to... So, I am going to solve the problem in x direction, I am going to solve the problem in y direction and somehow, combine the results. But then, what could happen is that I may over count; I may I may just over spend a lot of time figuring out, let us say, in the y direction, you know, this was bounded here, I will still in the x direction, I will report all these points and then I have to find the intersection, but in the mean time I have already done the work of identifying all those points, where I am reporting those points.

So, doing this, solving the problem somewhat in uncorrelated manner, will not get us the optimal results; we will use that. So, we cannot afford to ignore one axis and solve it independently on the two axes; that is what we are going to work, but there should be some way of combining the two surfaces; that is that is about (( )).

So, even before I take up this problem, let me discuss another thing. So, for the moment I will actually ignore that I have this boundary. And so, we are back to the minus infinity plus infinity and we are back to the one dimensional problem, but I will actually now view the one dimensional problem slightly differently.

So, if you recall, I mean I started by saying or at least introducing the range searching mechanism as partitioning points in a certain way

(Refer Slide Time: 32:18)



1. We need not examine R_7 because $R_7 \cap \Delta = \phi$

2. All points in R_3 should be reported. Because $R_3 \cap \Delta = R_3$

3. For the remaining we need to "examine further"

and then, if you find out, you know, and we build a tree on these partitions and then this range, with this range we find out, you know, which of these partitions will intersect partially; in which case, recursively search; if it does not intersect, we do not search; if it completely contains inside, we report the points. But, when we did the one dimensional solution we were not really using this this structure, right; we did not have this structure in mind, you know, that was a natural solution; you knew about binary search, find the left end points, keep walking till you head the right point and be done with it.

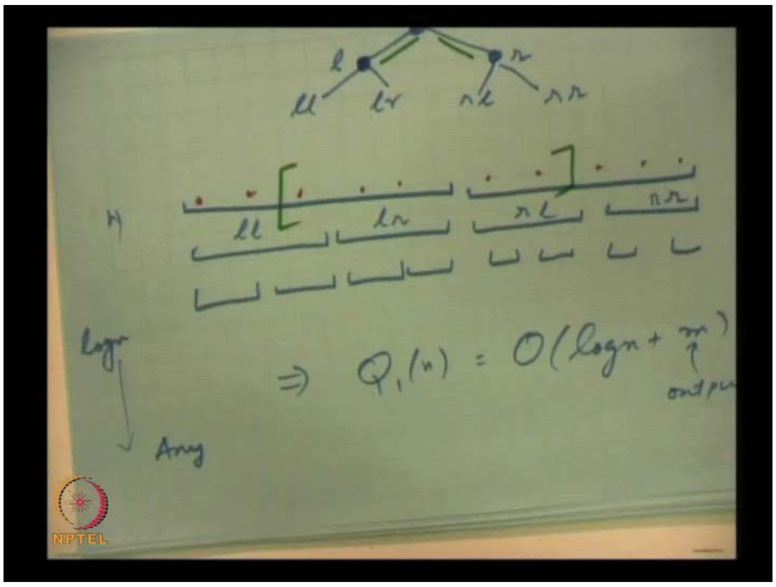
So, we certainly did not think in terms many kind of partitioning of the points and in many contexts, these partitionings, you know, they are some special partitioning like like we did in the case of k-d trees; we defined a special kind of partitioning based on the median and (( )), right.

So, we actually came up with a family of something like order n rectangular regions, so, think about it. The first time we partitioned, we got two rectangles, then n plus four. So, you know, over… So, so as we as we go down move down the tree, you know, it is a it is a, you know, it is a geometric series, two plus four plus eight, you know, eventually you cannot have more than n rectangles.

So, you actually, from a set of n points, we define a set of about order n partitions and these are often referred to as, you know, when there is a special kind of partitioning, they are called canonical partitions. So, when this when we devise the solution, these partitions are not some arbitrary partitions or arbitrary trusting; surely, you know it must have something to do with the efficiency that we expect out of it, you know, how how efficient this the query time should be? you know, how efficient the search should be and so on and sorry, the space should be and so on and so forth.

So, they are not arbitrary partitions. So, they are some special kind of partitions that are cleverly designed. Like in the case of k-d trees, we designed is rectangular partitions. So, now, let us go back to the one dimensional problem; we did not solve the problem using any kind of this partitioning technique.

(Refer Slide Time: 34:30)



So, let us revisit that problem. So, what what kind of partitioning you think would be appropriate for this? If you want to solve it like a general range query problem; ranges are intervals. So, fine! That is very natural one. Good! Right.

So, may be at the first level, you partition like this. So, we have a node that corresponds to the entire set of points and then we have these two nodes that corresponds to the left half and this to the right half, right. And then, you continue like that; you again find the median of this; you just continue that, right; and you have about depth of log n. Each node again is associated with a partition, right.

So, this at level one, there are two partitions. So, left and right and then again, maybe call it l l and l r and r l and r r, something, right. So, then you have these l l l r r l r r . So, you can have label each of these points and say again if this one is associated with this left set of n over 2 points n over 2 points. So, this becomes your is very becomes basically a one-d tree. So, k-d tree, we started with a two-d tree actually, without even bothering about the one-d tree. But this essentially is the one-d tree of the k-d tree.

So, you are going to again look upon this search process as you know, given any arbitrary interval given any arbitrary interval, what you do? You find out, you know, is this interval basically intersect a has a non empty intersection with this half and this half? In that case, if you see the along both like here and so on and so forth. So, can you quickly analyze how many nodes will end up (()) this one-d tree?

So, I am I have modified my one dimensional search, right. I have built a one-d tree which is the one dimensional k-d tree, right and I am searching using the same principle; I given any arbitrary interval; I will see whether it has non empty intersection, they I am sorry. It is completely overlaps, then of course, you know, I do not have to search for that, I can report all the points; if it has no intersection, I do not go down that branch; if it has partial intersection, I have to go down both branches.

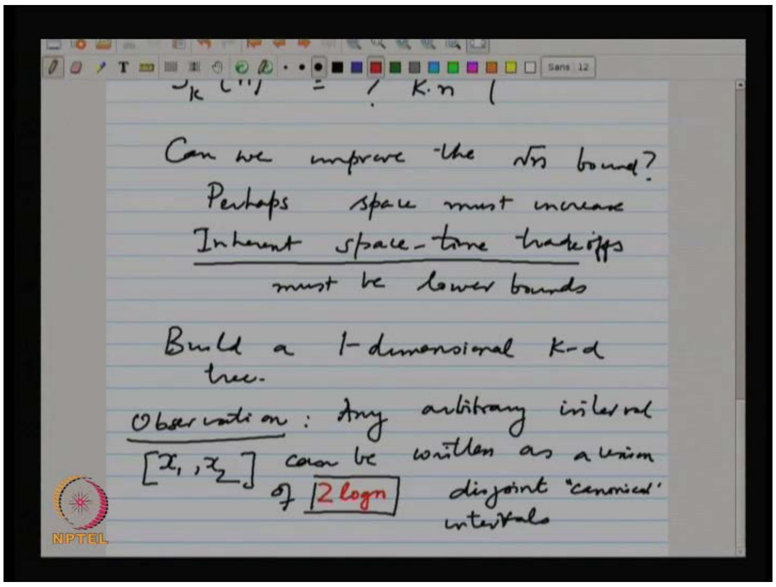
That is a great improvement, right. From square root of n, you are 2 log n. Why could you guess that? Why could you conclude that so quickly?

Because it has some similarity to that segment tree is that we did before, right. So, you can argue that at the forking node, after that you have some very ordered sequence of nodes that you are going to visit and at each level you are not going to have more than two nodes, basically that you are going to visit, right.

So, your overall numbers in a search time is actually just log n and of course, the reporting time is whatever the number of output points are there. So, this gives immediately, this leads to a query time of order log n plus output size, right. So, the one-d tree is actually quite efficient.

So, why are we doing all these? Well! Let us not go to k dimensional; we are going to look at only two dimensions. So, remember this picture. So, finally, what is happening? One one thing that we are getting out of all these is that any interval, any arbitrary interval, basically can be expressed as a union of, that is basically what what we have done here. So, this interval, you know, any arbitrary, so, any arbitrary interval, maybe I should write it down.

(Refer Slide Time: 39:12)



$S_k(n) = ?\ k \cdot n$ (

Can we improve the $\sqrt{n}$ bound?

Perhaps space must increase

Inherent space-time tradeoffs must be lower bounds

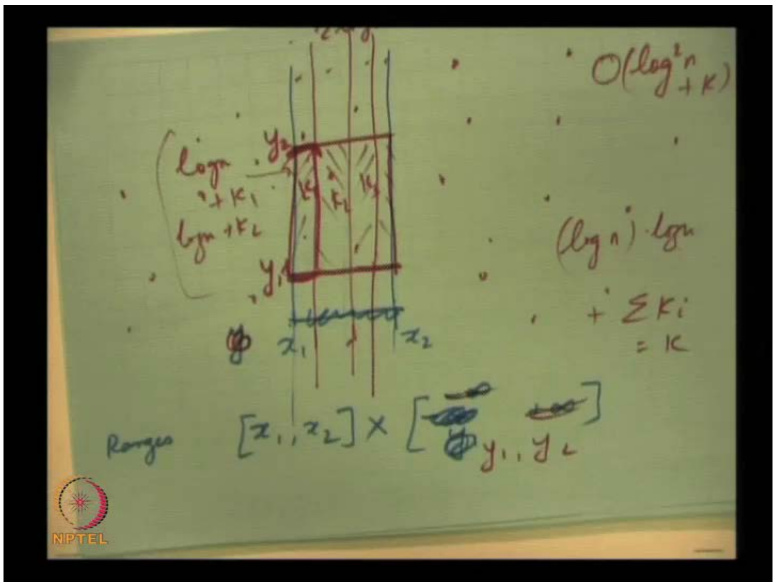
Build a 1-dimensional k-d tree.

Observation: Any arbitrary interval $[x_1, x_2]$ can be written as a union of $\boxed{2 \log n}$ disjoint "canonical" intervals

So, build a one dimensional k-d tree and observation is that any arbitrary interval, x 1 x 2 can be written as a union of, I will leave that blank, disjoint, where these are canonical intervals, right. So, I just mention this as canonical intervals. So, what is this figure? 2 log n.

So, let us go back to this picture.

(Refer Slide Time: 40:35)



Well! This is a one dimensional problem, but now we are viewing it as something like this particular interval x 1 x 2 can be expressed as union of some disjoint intervals, at most 2 log n intervals, right. So, maybe it is one interval is like this and other interval is like this, some disjoint intervals and these are a maximum of 2 log n, 2 log n canonical intervals. Fine! Now, what I do is I again go back, y 1 y 2.

How can we now, answer a query, a two dimensional query? Sort all the points in each interval and? Right. So, this two-d interval, now it is y 1 y 2. So, I have defined some y 1 y 2. So, let us say, this is y 1 and this is y 2. Again this entire region, can be expressed as union of this plus this plus this plus this disjoint intervals. Each of these intervals, the entire vertical slab contains many points that may not be inside the y 1 y 2 range, but then what I can do is within each of these intervals, I can build a data structure for one dimensional search along y direction.

(())

No no no. So, yeah. So, we are not doing it. So, what we have done is that we have done it very cleverly, right. We have actually not done it for all the points; we have done it for this canonical intervals. Yes. So, we have done it only for the canonical intervals.

Yeah! We will see we will see that. Sure. I mean that is a good point, but we will quickly come to that. Even before that, I hope that you are following how the query is going to look like?

So, I do the one dimensional query, I mean, it is like a one dimensional query, where you are finding out which canonical intervals which canonical intervals are spanned by the x 1 x 2? And that is the set of at most 2 log n disjoint intervals. Now, once we find out that these are the intervals, within each interval, I do a one dimensional search and each one dimensional search will cost me, log n plus the number of points reported, log n plus number of points reported.

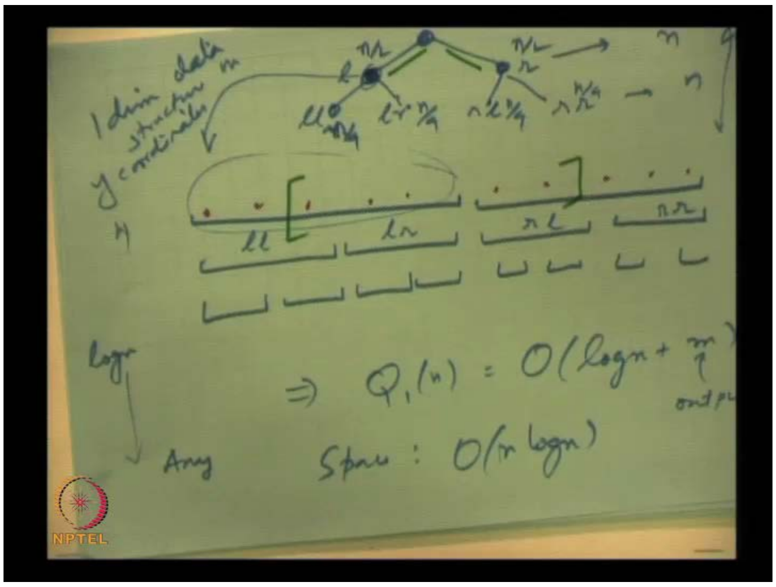
So, within this interval within this interval, now it is just a y 1 y 2 interval, whatever points are there, suppose there are k 1 points here, the k 2 points here and k 3 points here, this will cost me log n plus k k 1, this will cost me log n plus k 2; eventually, when you sum it up, what you get? You get basically log n, at most log n times plus basically, k I which is k. So, you get it? order log squared n plus k query bound.

So, query time, you immediately get an improvement from square root of n to log square of n, all the way you know; it is an exponential improvement actually, square root of n to log square of n. Of course, it is log square n, not log n; log n would have been even better.

So, we will struggle little bit more to get that. What is the space bound? Someone brought out a very good point. What is the space bound?

(()) Yes yes. So, yes yes, sure. So, let me put down, put up this picture again.

(Refer Slide Time: 45:19)



1 dim data structure y coordinates
n/2
n
ll
lr
rl
rr
n/4
log n
Any
$\Rightarrow Q_1(n) = O(\log n + m)$
out p
Space : $O(n \log n)$

So, this tree, the primary tree, there is a primary tree and a data structure actually within each node; let little bit more details. So, the primary tree is on the x points, it is the projection of the points on the x axis. But then, within each node, so, this node basically represents all these points; these points also have a x coordinate, which I have completely ignored here. So, within this, I am going to store the one dimensional data structure on y coordinates. That is linear time, right; the space will be linear space.

So, for all these n over two points… So, n over two points will be stored here, n over two points will be stored here, where there is a data structure. So, within each node, there is a secondary data structure for the searching in the y direction.

Similarly, for each of these nodes, there is a data structure on n over four points on the y direction. Now, when you add it up, you see that every level has basically a size n. So, it is blowing up. So, space is n log n. So, we are grown up the space.

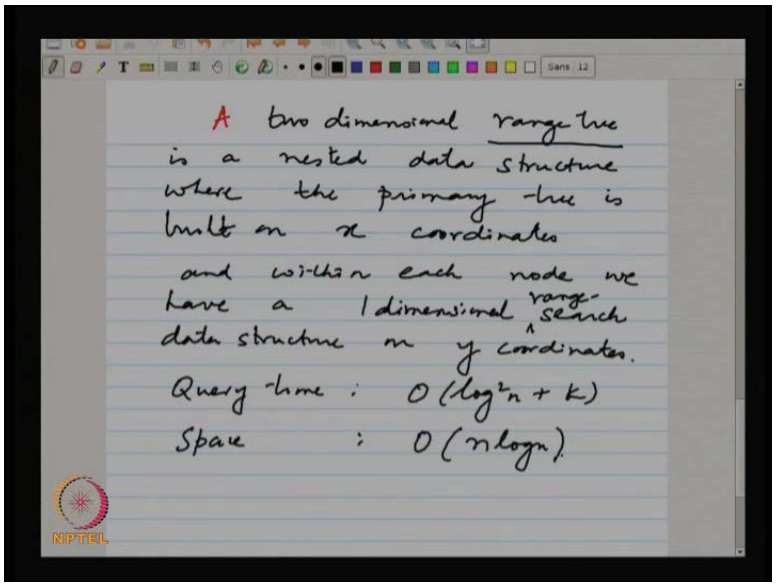
Does not look very serious; I mean a logarithmic factor at the and what you are getting benefited by is from square root of n, you have come down to log square of n, right.

So, it does not appear serious for theoreticians, but you know, the real world people they will really scream; no way, you know, I am storing giga bytes. I am not going to store, you know, another, whatever thirty times this; that is what comes down, right; log of n giga bytes.

So, for them, 30 n; for anyone, you know, even a factor of two or three is something that is sometimes not sustainable because, you know, your databases are really large and why should I use so much? Because space is there; the space, we have to live with the space, you know, we cannot is not pre processing time that we encounter once and forget.

So, it face certainly an issue. So, we will see what to do about that? Maybe, I will just give you some more thoughts. So, this was about and the... So, this data structure, where I have a primary data structure on the x coordinates and I have a nested data structure in this. So, this is basically the range tree. So, this is a range tree. So, I will just write it here. So, range tree, say two dimensional range tree, let us say

(Refer Slide Time: 48:07)



A two dimensional range tree is a nested data structure where the primary tree is built on x coordinates and within each node we have a 1 dimensional range-search data structure on y coordinates.

Query time : $O(\log^2 n + k)$

Space : $O(n \log n)$

where the primary tree is built on x coordinates and within each node we have and here we just observed that query time is order log square n plus k space n log n.

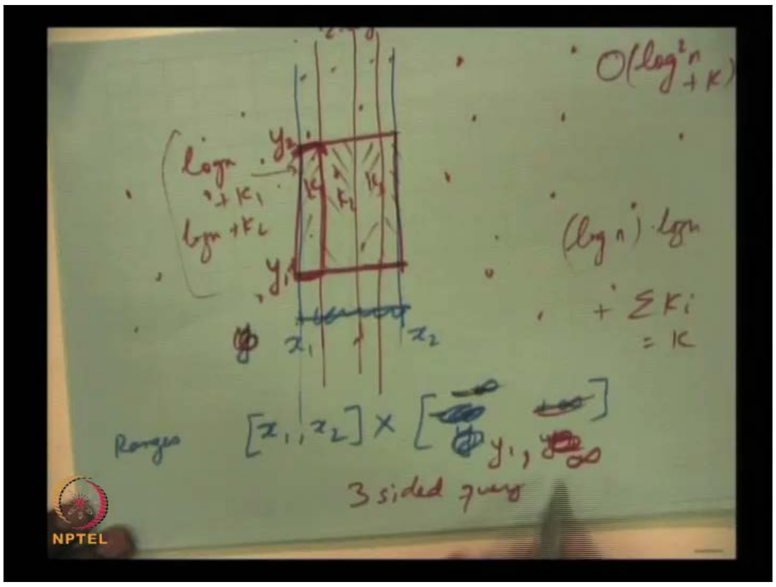
What happens, you know, if you go to higher dimensions? Can you extend this scheme.? So, suppose I got 3 dimensions. Now, I have a box; I build a primary data structure, let us say on. So, what is this? You know, so, y x z or something like that right. So, I build something on the z, the primary data structure. Again, there are only 2 log n canonical intervals. Now each of these canonical intervals is what? Is a is a two dimensional data structure.

So, inside the node, there will be a two dimensional data structure, right and so, this extends to any dimension; of course, you have to analyze what kind of query time will be there? and what kind of space requirement will be there? But in principle, this is a simple extension. But, you are going to certainly pay in terms of something in space and something in query time; it is not going to be log square n; it will be higher.

Yeah! That is a good case, I guess in k dimensional (( )) close to about log to the power k and query and about maybe a log to the power k n extra in space, but one has to rigorously prove that, right. So, this is about range trees and high dimensional range trees.

Coming back to this problem again. So, I have scratching it lots of time.

(Refer Slide Time: 51:06)



So, my final thing is again y 1 comma infinity, three sided. So, once I, sorry, just a two dimensional one dimensional range query was log n time, two dimensional range query, at least in the way that we have described it today, it is log square n. Will life be a simpler with a three sided query? This is called a three sided query. So, this is something that we will take up in the next class.